Implementation of Process Networks in Java

Richard S, Stevens[1], Marlene Wan, Peggy Laramie,
Thomas M. Parks, Edward A. Lee
DRAFT: 10 July 1997

Abstract

A process network, as described by G. Kahn, is a network of sequential processes connected by FIFO queues. Process networks, a generalization of dataflow graphs, are used extensively for representing signal processing algorithms. The requirement to run for long times with limited memory raises concerns about deadlocking and memory requirements. T. Parks gives an algorithm for executing a given process network forever in bounded memory, whenever possible. This algorithm depends on recognition of and response to deadlock conditions. We implemented this algorithm in Java and devised a new robust method for detecting deadlocks.

1.0 Introduction

Managing concurrency has become a critical issue in many domain-specific models of computation. Concurrency is required in reactive systems, which need to interact with an environment that produces multiple simultaneous stimuli. Networked applications are inherently concurrent, as is any application with a non-trivial user interface. The Java language provides threads, which are concurrent sequential programs that can share data, precisely to deal with such applications. However, programming with threads in their raw form can easily lead to errors that are difficult to diagnose. In particular, the Java language (correctly) does not define precisely how threads are scheduled. This is dependent on the implementation. Consequently, writing multithreaded applications that behave identically across multiple implementations requires painstaking care and attention to detail.

A *Kahn process network* is a directed graph, comprising a set of nodes (processes) connected by a set of directed arcs (FIFO queues) [1, 2]. Each process executes a sequential program. At any given moment this process may read a data token from one of its input queues, or it may write a data token to one of its output queues, subject to the following constraint: if a process attempts to read a token from an empty queue, the read is blocked, and the process must wait until a token is available on that queue. Such process networks are known to be determinate[2], which means that the sequence of tokens passing through

1. Richard S. Stevens is an employee of the U.S. Government, whose written work is not subject to copyright. His contribution to this work falls within the scope of 17 U.S.C. A7 105.
2. Kahn [1] gives a more general condition for determinacy, that processes be monotonic functions mapping input sequences to output sequences, where “monotonic” is with respect to a prefix order. In this paper, we will be concerned only with a subset of monotonic functions: those that can be described as sequential processes with blocking reads.

each queue over time is dependent only on the process network and not on its implementation [1, 2, 3]. A deadlock occurs when every process in the network is blocked while attempting to read an empty queue.

Kahn process networks provide a higher-level concurrency model that is inherently determinate, guaranteeing consistent behavior across implementations. Moreover, although process networks are not an ideal model of computation for all applications, they match a wide variety of applications well. They are excellent for computation-intensive real-time applications such as signal and image processing, as evidenced by the widespread acceptance of dataflow in the signal processing community. Dataflow is a special case of Kahn process networks [5]. Process networks are more difficult to use, however, for transaction-based applications where sharing a common database is key.

Many real-time applications are intended to run indefinitely using a limited amount of memory. This motivates our interest in process networks that (a) will run forever and (b) will do so using a bounded amount of memory. Some process networks can be analyzed statically to determine whether or not they meet these criteria. However, one of the authors (Parks) has observed that, in general, the questions of whether a process network meets these criteria are not decidable in finite time [6].

A further question arises about algorithms for scheduling the execution of a process network, and whether a given algorithm will use unbounded memory when bounded memory will suffice. Parks provides a method of running a process network forever in bounded memory, if possible. A capacity is assigned to each queue, and a write to a full queue is blocked. When a deadlock occurs involving some processes that are blocked while attempting to write, queue capacities are increased to break the deadlock, and processing continues.

We use the Java language to implement process network execution with Parks' algorithm. There is a thread for each process in addition to a main thread. The main thread creates all of the queues, creates and starts all of the process threads, and handles the deadlocks as they occur.

In the following discussion, we use pseudo-code to show the various methods. Section 2 discusses the details of process networks, with specific emphasis on how we implemented blocking reads. In Section 3 we give a precise description of blocking writes and Parks' prescription for running in bounded memory. Section 4 presents our Java implementation, together with our analysis of the two methods for detecting deadlocks. Section 5 gives some examples of process networks taken from [6]. Having run these examples using our implementation, we make some observations about the behavior that we observed.

2.0 Process Networks

A *process network* is a set of sequential processes communicating via first-in-first-out (FIFO) channels, or queues [1,2]. The following are some basic characteristics of a process network:

- Each process is a sequential, imperative program that reads data from its input queues and writes data to its output queues.
- Each queue has just one source and one destination.
- The network has no global data.
- Each process is blocked if it tries to read a communication channel with insufficient data. The read may proceed when the channel acquires sufficient data. Writing is not blocking; each queue may store an unlimited amount of data.

Kahn proved that a process network is *determinate* in the following sense: The sequence of tokens on each queue is determined solely by the process network and not by its implementation [1, 2]. It can also be shown that if a process network will deadlock, then it must deadlock in a unique state that is independent of the scheduling method [1, 2, 3].

Blocking reads may be specified in terms of `get` and `put` methods on the data queue. The `get` method is called by the process reading from the queue, and the `put` method is called by the process writing to the queue.

For example, assume that one token is transferred in each `get` and each `put` method. The `get` and `put` methods are defined in Figure 2.1.

```
int get() {
    if (empty) waitForPut();
    return firstToken();
}
void put (int value) {
    enqueue(value);
    if (waitingForPut) resumeGet();
}
```

Figure 2.1:
Blocking read implemented by `get` and `put` methods.

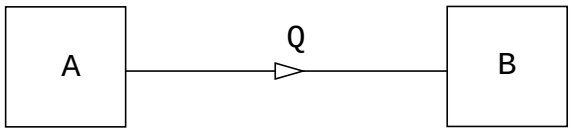


Figure 2.2:
A Process Network with two processes connected by a queue.

To see how this works, visualize a queue `Q` that connects two processes `A` and `B`, where `A` writes to `Q` and `B` reads from `Q`, as in Figure 2.2. Suppose that `Q` has one token and that `B` reads from `Q` by calling the `get` method on `Q`. The `get` first checks whether `Q` is empty. In this case, `Q` is not empty. The `firstToken()` method removes the first token from the

queue and returns its value. Now suppose that `B` reads another token from `Q`. This time, `Q` is empty, so `waitForPut()` is called, blocking the read. Suppose now that `A` writes to `Q` by calling the `put` method. This adds a new token to `Q`. In Java, one thread can *notify* another of an event that affects it. The `resumeGet` performs this notification by causing the corresponding `get` method to be resumed after the `put` is complete. At this point `Q` is empty but not blocking a read. If `A` now writes to `Q`, a token will be added to `Q`. Because `B` is not waiting for a token, notification is not necessary.

3.0 Bounded Memory Scheduling

In a process network [1, 2], no restriction is placed on the number of processes in the network or on the amount of memory required by individual processes. Either or both may be unbounded. A process network may require unbounded memory for any one or a combination of the following reasons:

- One or more queues must store an unbounded number of tokens.
- The tokens have unbounded size.
- One or more processes in the network requires unbounded memory.
- While running, the process network is reconfigured by the continual addition of new processes and/or new queues.

Henceforth, unless otherwise stated, we take *bounded memory scheduling* to mean that the number of tokens on each queue should remain bounded. We are not concerned about token size, memory requirements of individual processes, or the number of processes and queues in the process network.

We summarize the results of Parks [6]. The following questions arise:

- Will a given process network run forever without reaching a deadlock state?
- Will a given process network run in *bounded memory*? Specifically, is there a bound B and a scheduling scheme for running the process network such that the number of tokens on each queue never exceeds B?

Each of the above questions is equivalent to the halting problem of a Turing machine and is thus undecidable [4, 6]. If a process network eventually reaches a deadlock state, then it runs in bounded memory. Thus the question of undecidability for execution in bounded memory applies in the case where running forever is either possible or not known.

At any given moment, several of the processes may be able to run. If the process network is running on a system with a single processor, then a *scheduling policy* may be used to interleave the execution of the processes. Several scheduling policies are well known, such as *data driven* and *demand driven*. Data driven scheduling activates a process as soon as sufficient input tokens are available. Demand driven scheduling defers process activation until its output tokens are needed. For further information about these and other scheduling policies, see [6].

Most traditional scheduling policies, such as data driven and demand driven, will execute process networks forever, if possible. For each of the traditional scheduling policies there is an example of a process network which the policy runs, using unbounded memory when bounded memory would suffice [6].

Parks gives an algorithm that runs a process network forever in bounded memory whenever that is possible. If a process network requires unbounded memory to run forever, then there is a conflict between the two goals of running forever and doing so in bounded memory. In this case Parks' algorithm prefers the use of unbounded memory to run forever over terminating to stay within bounded memory.

Parks' algorithm assigns a *capacity* to each queue in the process network, which limits the number of tokens that the queue can contain. Just as a read is blocked when there is insufficient data, a write is blocked when there is insufficient capacity.

With the introduction of blocking writes to process networks, the questions of determinism and deadlock must be addressed anew. By similar arguments it can be shown that such a process network is determinate, and that if it will deadlock, then it must deadlock in a unique state.

```
int get() {
    if (empty) waitForPut();
    if (waitingForGet) resumePut();
    return firstToken();
}
void put (int value) {
    if (full) waitForGet();
    enqueue(value);
    if (waitingForPut) resumeGet();
}
```

Figure 3.1: Blocking writes and blocking reads implemented by `get` and `put` methods.

For blocking writes, modified `get` and `put` methods are shown in Figure 3.1, which retain blocking reads. Referring again to Figure 2.2, suppose that `Q` has a capacity of 1 and that `Q` has a token. Then `Q` is full. If `A` writes to `Q` by calling `put`, `waitForGet()` is called, blocking the write. Now suppose that `B` reads from `Q` by calling the `get` method. A token being available, there is no need to wait. The `resumePut` causes the corresponding `put` method to be resumed after the `get` is complete. The `firstToken()` method removes the first token from the queue and returns its value.

With blocking writes and blocking reads, a deadlock may now occur in which one or more processes are blocked while writing. This is an *artificial deadlock*. A *true deadlock* occurs when all processes are blocked while reading.

To run in bounded memory when it is possible to do so, we may use any scheduling policy that ensures eventual execution of an executable process (e.g., data driven or demand driven) until a deadlock occurs. If an artificial deadlock occurs, we increase the capacities of the queues to break the deadlock and continue running. One of the following cases must apply:

- The process network can run forever in bounded memory: If the initial capacities are sufficiently large, the process network will run forever without a deadlock ever occurring. Otherwise some artificial deadlocks will occur, and the queue capacities will be increased. Eventually the capacities will become sufficiently large for the process network to run without further deadlocks. The process network will run forever in bounded memory.
- The process network can run forever but requires unbounded memory: Artificial deadlocks will occur continually, and queue capacities will be increased without bound. The process network will run forever using unbounded memory.
- The process network eventually halts: In this case, a true deadlock will occur, possibly after some artificial deadlocks and resulting increases in queue capacities.

Figure 3.2 shows this scheduling algorithm in pseudo-code.

```
void deadlockManager() {
  do {
    waitForDeadlock();
    if (trueDeadlock) terminate();
    increaseQueueCapacities(); // Artificial deadlock
  } forever;
}
```

Figure 3.2:
Scheduling algorithm for execution in bounded memory, if possible.

When an artificial deadlock occurs, there is some latitude in selecting the queues whose capacities should be increased. For example, increasing the capacity of every queue will achieve the ultimate goal of executing in bounded memory whenever possible. With an eye toward memory conservation, this is unnecessary. A better choice is to increase the capacities of those queues that are blocking writes, because nothing will be gained by increasing the capacities of queues that are only blocking reads. Beyond that, Parks observes that it is sufficient to increase the capacity of just one queue, i.e., one with minimum capacity chosen from the queues that are blocking writes [6].

4.0 Implementation in Java

The Java language [7, 8, 9] has a number of features that makes programming and debugging relatively easy:

- Java is object oriented.
- Java supports multiple threads.
- Java supports exception handling.
- Java is strongly typed.
- Java collects garbage, eliminating memory leaks.
- Java provides run-time checks (e.g., array index out of bounds).

In our implementation, we define a thread for each process in the process network. Thus, instead of controlling the processing with a data-driven, demand-driven, or other traditional scheduling policy, we run all of the process threads, using the `wait()` and `notify()` methods of Java threads to implement blocking reads and writes. In addition, we define a main thread to create the queues, to create and start the process threads, and (as we shall see) to detect and handle deadlocks.

When a deadlock occurs, suddenly nothing happens, because all process threads are waiting. Thus arises a problem: *how to detect a deadlock.* There are at least two possible solutions, both using a main thread to monitor the execution of the processes in the process network:

(1) Run the main thread at a lower priority than the process threads. When a deadlock occurs, no process threads are running, and so the main thread runs to analyze and handle the deadlock. An implementation of this solution is described in [6].

(2) Keep a count of all current read and write blocks. A deadlock is detected when the total number of blocks equals the number of processes, at which point the main thread analyzes and handles the deadlock. This is a new approach.

Solution (1) seems attractive for the following reasons:

- There is no run-time overhead except during a deadlock.
- Design and coding is relatively simple.

Solution (1) works if the implementation is run on a single processor. However, on a multi-processor network, this solution might allow the main thread to run on an idle processor while there are process threads still running.

Solution (2) requires some run-time overhead to keep track of the number of current read and write blocks. Each read block and write block is recorded by incrementing the respective counter. If a deadlock condition exists, then the main thread is resumed so that it can break the deadlock. Each read unblock and write unblock is recorded by decrementing the respective counter. This solution works on a multiprocessor environment as well as on a single processor. Judging this to be more robust, we choose it for our implementation.

The methods to support deadlock detection are shown in Figure 4.1. These methods are called from within the `get` and `put` methods. A process does not know when it is

blocked; it simply waits for the `get` or `put` to return. This approach is valid, because each process runs a sequential program; each call to a `get` or a `put` must complete before the process continues. Thus each process may be blocked only by one `get` or `put` at any given time. A deadlock occurs when all processes are blocked, which occurs exactly when the total number of read and write blocks equals the total number of processes.

We implement mutual exclusion for the block counters, which are global data shared by all the threads. This is accomplished in Java by declaring methods to be *synchronized*. When a thread calls a synchronized method of an object, that object is locked, preventing any other concurrent calls to synchronized methods of that object [7, 8, 9]. If a synchronized method waits for a condition, the lock is released until that condition is set and the method is resumed.

```
void recordReadBlock() {
    increment(readBlockCounter);
    deadlockTest();
}
void recordReadUnblock() {
    decrement(readBlockCounter);
}
void recordWriteBlock() {
    increment(writeBlockCounter);
    deadlockTest();
}
void recordWriteUnblock() {
    decrement(writeBlockCounter);
}
void deadlockTest() {
    if (readBlockCounter + writeBlockCounter
       == processCount) {  // deadlock detected
       resumeDeadlockManager();
    }
}
```

Figure 4.1: Methods to support block counting and deadlock detection.

Figure 4.2 shows the additional queue method to increment the queue's capacity.

```
void incrementCapacity () {
  increment(capacity);
  resumePut();
}
```

Figure 4.2:
Queue method for incrementing a queue's capacity.

Figure 4.3 shows the `get` and `put` methods in the Queue class modified to record read and write blocks.

```
int get() {
    if (empty) {
       recordReadBlock();
       waitForPut();
    }
    if (waitingForGet) {
       recordWriteUnblock();
       resumePut();
    }
    return firstToken();
}
void put (int value) {
    if (full) {
       recordWriteBlock();
       waitForGet();
    }
    enqueue(value);
    if (waitingForPut) {
       recordReadUnblock();
       resumeGet();
    }
}
```

Figure 4.3: The `get` and `put` methods with calls to record blocking and unblocking reads and writes.

For the main thread we use the algorithm shown in Figure 3.2, increasing the capacity of just one queue for each artificial deadlock, a blocking queue with the lowest capacity as suggested in [6].

5.0 Tests and examples

Figure 5.1 through 5.8 exhibit pseudo-code definitions for the processes used in the test cases. See [6] for details.

```
int stream W = process interleave (int stream U, int stream V) {
  do {
    put(get(U), W);
    put(get(V), W);
  } forever;
}
```

Figure 5.1: A process to interleaves two streams into one.

```
(int stream V, int stream W) = process alternate (int stream U) {
  do {
    put(get(U), V);
    put(get(U), W);
  } forever;
}
```

Figure 5.2: A process to distribute odd and even tokens from ones stream to two.

```
int stream V = process begin_with (int stream U, int x) {
  put (x, V);
  do {
    put(get(U), V);
  } forever;
}
```

Figure 5.3: A process to insert a token at the beginning of a stream.

```
(int stream V, int stream W) = process duplicate (int stream U) {
  do {
    int u = get(U);
    put(u, V);
    put(u, W);
  } forever;
}
```

Figure 5.4: A process to duplicate a stream.

```
int stream V = process add (int stream U, int x) {
  do {
    put(get(U) + x, V);
  } forever;
}
```

Figure 5.5: A process to add a constant to each token of a stream.

```
(int stream V, int stream W)
            = process multiple (int stream U, int y) {
  do {
    int u = get(U);
    if (u mod y == 0) put(u, V);
    else (put u, W);
  } forever;
}
```

Figure 5.6: A process to separate the multiples of a given constant from the non-multiples.

```
(int stream W) = process merge(int stream U, int stream V) {
  int u = get(U);
  int v = get(V);
  do {
    if (u < v) {
      put(u, W);
      u = get(U);
    }
    else if (u > v) {
      put(v, W);
      v = get(V);
    }
    else {
      put(u, W);
      u = get(U);
      v = get(V);
    }
  } forever;
}
```

Figure 5.7: A process to implement a sorting merge. Two monotonically increasing input streams are merged into one monotonically increasing stream. Two equal tokens on the two input streams result in one output token.

```
process print (int stream U) {
  do {
    print(get(U));
  } forever;
}
```

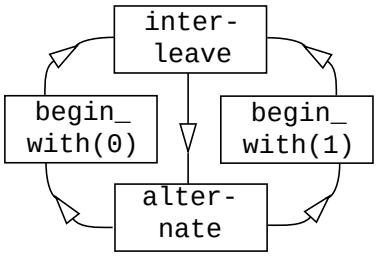
Figure 5.8: A process to print the tokens of a stream.

To study the behavior of various process networks using our implementation, we offer a choice of three modes of execution. We discuss the results of various test cases running under the different modes.

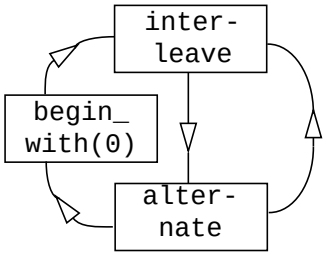
- The *free memory* mode supports execution without blocking writes. Whenever a `put` is called, if the queue is full, the capacity is increased immediately.
- The *fixed memory* mode never increases the capacity of any queue. Initial queue capacities are assigned, and the processes in the process network run until the first deadlock occurs (either artificial or true), at which point execution is terminated.
- The *bounded memory* mode supports execution forever in bounded memory, if possible, as described above.

Example 5.1 is a process network that will always execute forever in bounded memory regardless of the scheduling policy. Example 5.2 is a process network that will eventually

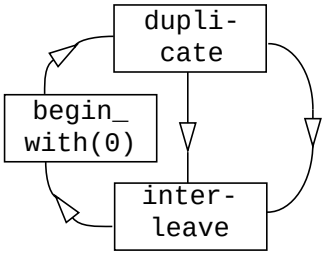
become deadlocked with any scheduling policy. Example 5.3 will run forever, but it requires unbounded memory to do so. Running these process networks with the various modes, we observe the expected behavior.



Example 5.1

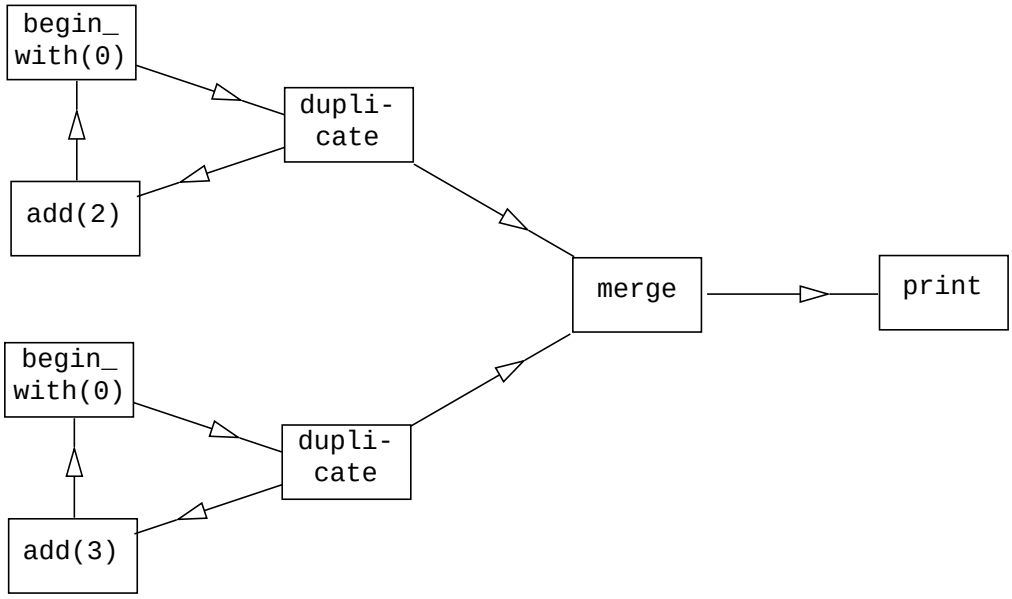


Example 5.2



Example 5.3

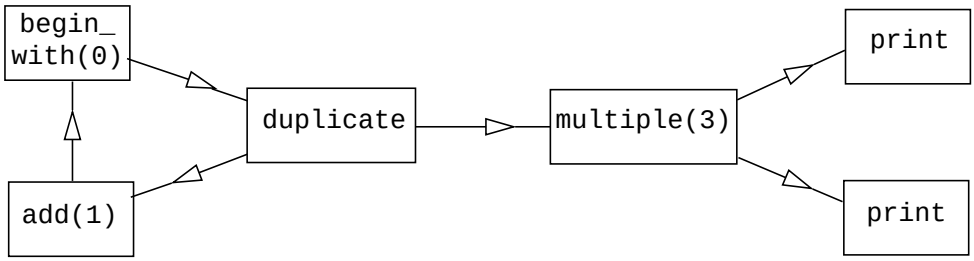
In example 5.4, each of the directed cycles is a source that produces a linearly increasing sequence starting with 0. The upper source increments by 2; the lower source increments by 3. These two sequences are merged at the process `merge` to produce a single increasing sequence that is printed by `print`. A data driven scheduler will cause the output channel from the lower source to grow in size over time, thus using unbounded memory. A demand driven scheduler will run this process network in bounded memory. With the free memory option, our implementation runs this example with the output channel from the lower source growing as we would expect with a data driven scheduler. With the bounded memory and fixed memory options, this example runs forever with the capacity of 1 for every channel.



Example 5.4

In example 5.5, the process `multiple(3)` outputs the multiples of 3 to its upper output queue and the non-multiples of 3 to its lower output queue. A data driven scheduler will run this example in bounded memory. A demand driven scheduler will fail to run in

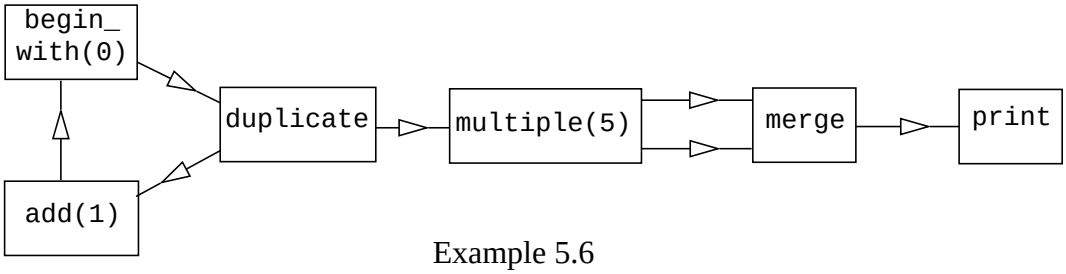
bounded memory, because the two sink processes `print` must execute at different rates. When run with the free memory and bounded memory options, no queue ever contains more than one token.



Example 5.5

Example 5.6 illustrates a graph in which different queues have different bounds. The source at the left produces the sequence 0, 1, 2, ... The process `multiple(5)` outputs all multiples of 5 to its upper output queue and all non-multiples of 5 to its lower output queue. The process `merge` then merges these two sequences in ascending order, causing the process `print` to print the original sequence 0, 1, 2, ...

This example demonstrates that the bounded memory scheme only guarantees bounded memory, not minimum memory. In fact, the free memory option uses less memory than the bounded memory option. To run forever, the lower output queue from `multiple(5)` must be able to contain three tokens; for all other queues, a capacity of one token is sufficient. The free memory option runs this process network within those limits.



Example 5.6

Running with the bounded memory option and all queue capacities initially set to 1, a deadlock occurs with `duplicate` and `multiple(5)` write blocked: `duplicate` is blocked by its only output queue, and `multiple(5)` is write blocked by its lower output queue. All other processes are read blocked. The two write blocking queues have capacity 1. Incrementing the capacity of one of these two write blocking queues from 1 to 2 breaks the deadlock, and another deadlock occurs with the same two processes blocked by the same two queues. This time one of the two queues has capacity 2, and the other has capacity 1, which is incremented to 2. The process repeats until the lower output queue of `multiple(5)` has capacity 3, at which point the process network has sufficient memory to run forever. This behavior is consistent with observations reported in [6].

6.0 Summary and Conclusions

We discussed an implementation of process networks in Java. Instead of designing a scheduling algorithm that decides which process to run, we implement a policy that uses multiple threads with blocking reads and works correctly regardless of the scheduling algorithm used for the threads. The only requirement we make of such of a scheduling algorithm is that if there is a process that is not blocked, then at least one process will run.

To ensure that memory usage is bounded whenever possible, we implement queue capacities with blocking writes and increase the capacity of a selected queue whenever an artificial deadlock occurs. To detect deadlocks, we track the number of blocked processes, a deadlock occurring if and only if this number is equal to the total number of processes in the network. In this way we recognize deadlocks, determine the type of deadlock (artificial or true), and respond accordingly. This is a valid approach on a multi-processor distributed system as well as on a single processor with threads.

7.0 References

[1] G. Kahn, *The semantics of a simple language for parallel programming*, Information Processing 74, pp. 471-475, Stockholm, August 1974.

[2] G. Kahn & D. MacQueen, *Coroutines and Networks of Parallel Processes*, Information Processing 77, pp.993-998 Toronto, August 1977.

[3] R. Stevens & D. Kaplan, *Determinacy of Generalized Schema*, IEEE Trans. Comp., Vol. 41 pp. 776-779, June 1992.

[4] J. Buck, *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model*, Ph.D. Thesis, University of California, Berkeley, 1993.

[5] E. Lee & T. Parks, *Dataflow Process Networks*, IEEE Proceedings, pp. 773-799, May 1995

[6] T. Parks, *Bounded Scheduling of Process Networks*, Ph.D Thesis, University of California, Berkeley, 1995.

[7] K. Arnold & J. Gosling, *The Java Programming Language*, 1996, Addison Wesley.

[8] G. Cornell & C. Horstmann, *Core Java*, 1996, Prentice Hall.

[9] M. Grand, *Java Language Reference*, 1997, O'Reilly.